SABRE Security

RE: We need better tools !

Halvar Flake – DefCon 2006 – Las Vegas

Introduction

Welcome to my talk. This talk ...

- ... is different from pretty much all my other talks I've given
- ... does not contain any cool bugs (I am getting too old and lame for that)
- ... does not present solutions to problems, really
- ... is to one part a 'wish-list' of things I'd like to have and problems I would like to see solved
- ... is to a second part an 'idea-dump' – for all the problems discussed here, there will be a few vague and probably incorrect ideas for their solution
- ... is to an extent a set of 'challenges' for people that want to advance the state of reverse engineering

Engineering

What really is a broad definition of „reverse engineering“ ?
Before we start understanding **reverse engineering**, let's first understand **engineering**, or specifically **software engineering**:

- A problem is defined
- A system is designed, consisting of multiple components and their interaction
- The components are constructed / acquired / stolen
- The components are integrated

Generally, the written source code is the last step of a '**concretisation**' of an **abstract design**.

Engineering

The engineering process does not only produce the final product, but also a high-level design (implicit or explicit), interfaces, trust relationships between components, (some) documentation etc. etc.

In practice, the engineering process is a lot messier than described above.

This does not change the fact that the existing code **defines implicit design, implicit modularization, implicit trust relationships between components.**

Reverse Engineering

What really is a broad definition of „reverse engineering“ ?
Let's first understand what it is **not**:

- **Disassembly** is not the end goal of reverse engineering. While a clean disassembly is the starting point for almost any reverse engineering ventures, the clean disassembly is not the end goal, but the first step of a journey.

  A good disassembly recovers all the 'functions' and groups them accordingly, and recovers all directly resolvable subfunction calls

Reverse Engineering

What really is a broad definition of „reverse engineering“ ?
Let's first understand what it is **not**:

- **Decompilation to source** is not the end goal of reverse engineering.

  While a decompilation to source will allow people that are not fluent in their reading of assembly to read the program, it does not necessarily help program understanding much.

  Having source code available is very different from understanding the source code. Having all parts of a car and their assembly instructions does not recover the reasoning behind the design of each part.

Reverse Engineering

What really is a broad definition of „reverse engineering“ ?
Let's first understand what it is **not**:

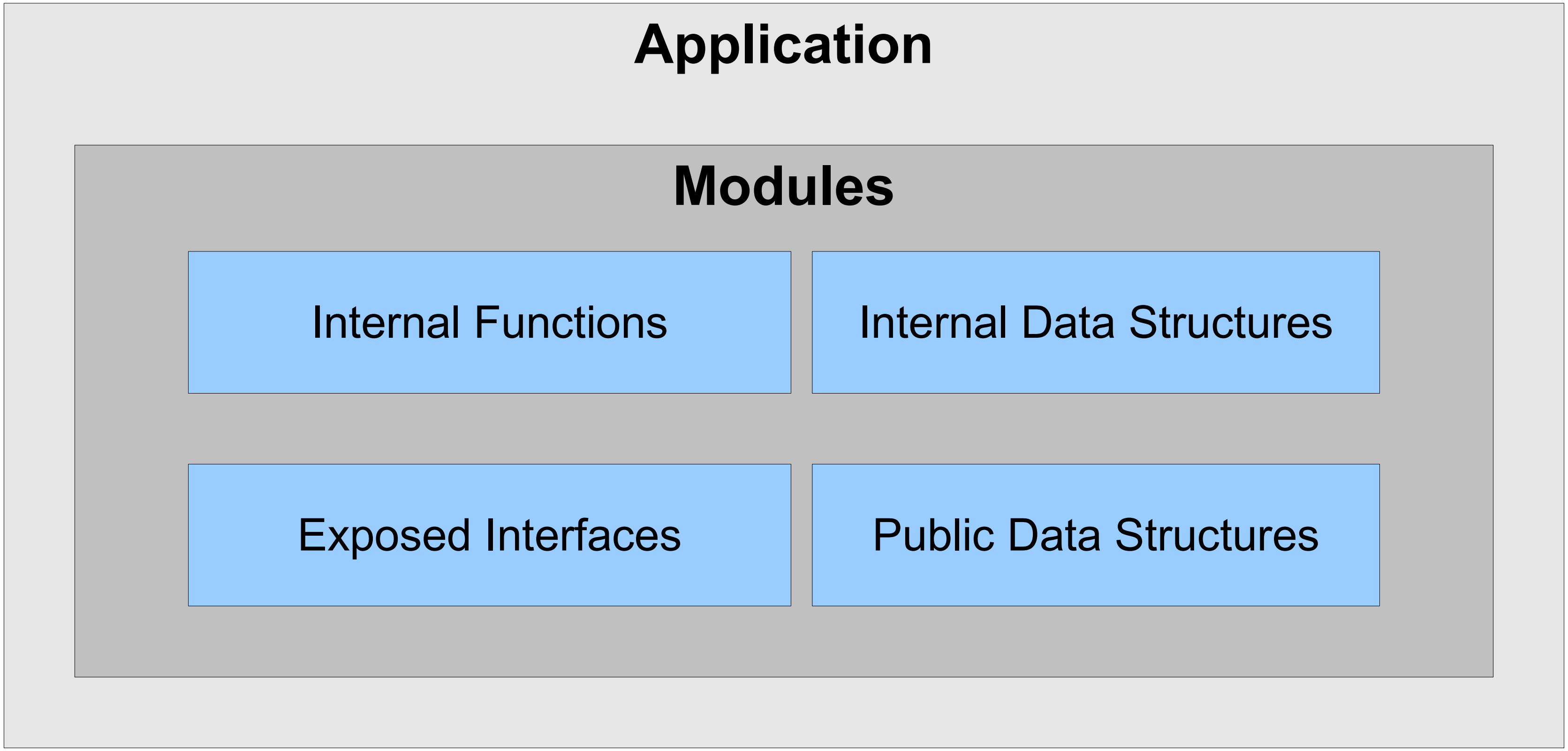
- **Recovery of high-level abstractions and program understanding** is the end goal of reverse engineering.

  In addition to recovering the pure language constructs, we want to recover higher-level abstractions: Modularization, Interfaces etc.

  (Yeah, bug-finding is the goal most of the time, but that's a subset of the above)

Software

So what does software really consist of ? Difficult question in the „non-OOP-paradigm“. One possible abstraction:



Application

Modules

Internal Functions

Internal Data Structures

Exposed Interfaces

Public Data Structures

Software

So, in „non-OOP-paradigm“-software, can we recover these abstractions ?

Let's review some design principles (from Dowd/McDonald):

- Loose Coupling (no, not that):
  Modules should communicate through few, well-defined „public“ functions which exchange limited amounts of data that are sanitized.

- Strong Cohesion
  A module should consist of functions performing strongly related tasks.

Challenge #0

Reconstruction and enumeration of possible return values from functions.

SABRE Security

Challenge #1

Full-executable data structure reconstruction:

Given an executable, reconstruct all data structures that are used in the executable.

Reconstruct all relations between the members of these data structures:

strucA.memberA points to strucB etc.

Construct a graph from these relations where every data structure is a graph, and pointers between data structures are edges.

This graph can be used to identify recursive data structures such as linked lists and trees.

Challenge #2

Full-executable object reconstruction:

Given an executable, reconstruct all class data structures used in the executable.

Associate the classes with their methods and reconstruct the inheritance hierarchy between the classes.

Create UML diagrams from the executable.

Merge the UML diagram of the classes with the typeinfo generated in Challenge #1.

Challenge #3

Automated decomposition of executables into modules

Given an executable, group the functions into subsets that reflect both loose coupling and strong cohesion.

- Approach 1: Calculate dominator trees on the callgraph, decompose from there
- Approach 2: View this as an optimization problem to decompose the callgraph into strongly connected components by removing a minimal number of nodes
- Approach 3: Attempt to group functions not by call hierarchy but by usage of the same data structures

Group the functions in the subset into public and nonpublic functions, and the data structures into public and nonpublic data structures. public and nonpublic functions.

Challenge #4

Recovery of template-generated code from the executable

Challenge #5

Automating input crafting for an executable path.

Challenge #6

Automating the analysis of translation-and-emulation based obfuscators.

SABRE Security

Challenge #8

Construct a mode of callgraph visualisation that not only shows the calls-to-relationships but also the order of subfunction calls in a given functions.

Challenge #9

Build an automated system consisting of

- Static analyzer to find bugs
- Input generator to trigger bugs
- Exploitation automation to automatically build exploits